Automating Test Automation

Suresh Thummalapenta*, Saurabh Sinha*, Nimit Singhania*, and Satish Chandra†
*IBM Research - India
†IBM T. J. Watson Research Center

***Abstract*—Mention "test case", and it conjures up the image of a script or a program that exercises a system under test. In industrial practice, however, test cases often start out as steps described in natural language. These are essentially directions a human tester needs to follow to interact with an application, exercising a given scenario. Since tests need to be executed repeatedly, such *manual* tests then have to go through *test automation* to create scripts or programs out of them. Test automation can be expensive in programmer time.**

We describe a technique to automate test automation. The input to our technique is a sequence of steps written in natural language, and the output is a sequence of procedure calls with accompanying parameters that can drive the application without human intervention. The technique is based on looking at the natural language test steps as consisting of *segments* that describe actions on targets, except that there can be ambiguity in identifying segments, in identifying the action in a segment, as well as in the specification of the target of the action. The technique resolves this ambiguity by backtracking, until it can synthesize a successful sequence of calls.

We present an evaluation of our technique on professionally created manual test cases for two open-source web applications as well as a proprietary enterprise application. Our technique could automate over 82% of the steps contained in these test cases with no human intervention, indicating that the technique can reduce the cost of test automation quite effectively.

I. Introduction

A. What Is Test Automation?

Developers create unit test cases in the form of small programs, typically in a framework such as JUnit [1]. A suite of unit tests can be run as often as desired without any manual intervention.

The story for system tests, which are end-to-end tests for an application, is different. By application, here we mean a web application that resides on a server and is accessed via a web browser. System tests are typically written by testers, not the developers of the application; in fact, system testing is often carried out in a different organization altogether.

System testing starts with identification of test scenarios from requirements and use cases, and creation of *manual test cases* from those scenarios. A manual test case is a sequence of test steps written in natural language. Figure 1 shows a sample manual test case written by a professional tester.

Manual test cases are intended to be used by humans. Unlike unit tests, they do not have the benefit of efficiently repeatable test execution. Therefore, executable test scripts (or programs) need to be created from manual test cases, so that the suite of system tests can be run efficiently.

```
1: Launch the application through the link
     http://localhost:8080/store/Default.jsp
2: Enter the intended book search name as "MySQL" at the
     "Title" Edit field and select "Category" as "All"
     by Drop  down list and then Click "Search" Button
3: Select a title from the list of all Search Results
     displayed and then click either on the image of the
     book or on Name of the Book
4: Enter login guest and password guest, and click login
5: Enter the Quantity "1" and Click on
     "Add to Shopping Cart" Button
6: Verify whether Order # and Total are displayed
```

Figure 1. An example manual test case for an online BookStore application written by a professional tester.

Test automation is the task of creating a mechanically interpretable representation of a manual test case. Such a representation could be a program in a general-purpose programming or scripting language (*e.g.,* Java or VBScript). Alternatively, automation could be realized by formulating a test case as a sequence of *keywords* or calls to executable subroutines, along with the relevant arguments. This sequence can be interpreted automatically by a driver program. Figure 2 shows this keyword form for the test case of Figure 1. The second row, for example, is equivalent to a call *locateTextBox("Title").enter("MySQL")*, where *enter* applies to the text box identified by the first call. The last two rows in the table are verification steps, which ensure that the state of the browser is as expected. Yet another alternative is to record a tester's GUI interaction, while the tester performs a manual test, into a script that can be replayed later. These approaches entail different degrees of control over interaction with the application, and require different degrees of technical skill in carrying out test automation. Writing in Java is more complex, albeit more general than using keyword-driven automation. Record-replay requires no programming skill but offers least control; moreover, recorded scripts tend to be brittle—they can easily break in the presence of even minor application GUI changes.

The cost of test automation has tremendous implications on the cost of testing and the quality of the application being developed. The number of manual test cases for an enterprise-grade application typically is in the thousands. Given this scale, the cost of test automation adds up. Note that even keyword-based automation takes time and expertise. For example, the target column as shown in Figure 2 requires careful specification of the location of a GUI element in the document object model (DOM) as presented in a browser. Moreover, the tester has to compensate for imperfections in the manual test case. It is also the case

UI Element Type	Action	Target	Data
	goto		http://localhost:8080/ store/default.jsp
Text box	enter	Title	"MySQL"
Combo box	select	Category	all
Button	click	Search	
Link	select	MySQL & PHP From Scratch	
Text box	enter	Login	guest
Text box	enter	Password	guest
Button	click	Login	
Text box	enter	Quantity	1
Button	click	Add to Shopping Cart	
Label	exists	Order #	
Label	exists	Total	

Figure 2. A keyword-based script, consisting of a sequence of triples, where each triple specifies an action, the target of the action, and any required data.

that after application changes, some of the automated tests needs to be fixed. Consequently, only a fraction (sometimes less than 20%) of manual tests end up being automated, as projects are unwilling to pay for the cost of test automation. Typically, there is insufficient time to perform manually all the manual test cases in between releases of an application. This compromises on the quality of the application, which the testing phase was supposed to ensure. Therefore, it is important to try to reduce the cost of test automation.

B. *Automating Test Automation*

In this paper, we address the problem of creating automated test scripts from manual test cases, specifically, in the context of keyword-driven automation. We have built a tool, ATA, that takes as input a manual test case of the form shown in Figure 1 and outputs a script as shown in Figure 2, which is mechanically interpretable by a driver, also included with the tool. Thus, our tool (1) enables execution of test cases written in natural language, and (2) creates a representation suitable for automated execution.

Since natural-language understanding is still challenging for computers, our technique relies on basic parsing capability only. Rather, we rely on the observation that the style in which testers write manual steps have a predictable structure consisting of *segments*, where a segment is an instruction for the tester to perform an action on a user interface. Examination of over a thousand test steps taken from manual tests of real applications supports this observation. Moreover, the tests have a restricted vocabulary (*e.g.,* consisting of nouns that are pertinent for the application being tested and verbs that specify actions on user-interface elements); this is also borne out by our experiments (see Section IV-C).

Consider Figure 3, which illustrates the segments in the second step in the manual test case of Figure 1. Column 1 shows the test step with *conjunction* words highlighted. Column 2 shows preliminary segments obtained by breaking up the sentence at conjunction words. In each segment, we look for a *verb* to specify the action, a *noun* tagged as a direct object of the verb to specify the target of the action, and another *noun* (if present) to specify the associated data.

Test Step	Preliminary Segments	Candidate Tuples
Enter the intended book search name as MySQL at the Title Edit field **/and/** select Category as All **/by/** Drop down list **/and/** then Click Search Button	**Enter** the intended book search name as MySQL at the Title Edit field	1 <Enter, Name, Book>
		2 <Enter, Name, MySQL>
		3 <Enter, Name, Title>
		4 < ... >
	select Category as All	1 <Select, Category, All>
	Drop down list	1 <Drop, Down, List>
		2 <Drop, List, Down>
	then **Click** Search Button	1 <Click, Search , Button>
		2 <Click, Search, –>
		3 <Click, Button, Search>
		4 <Click, Button, –>

Figure 3. Illustration of the segmentation of step 2 of the sample test case.

This can be achieved by parts-of-speech (POS) tagging, a basic operation in a natural language parser, such as the Stanford Natural Language Parser [2]. Verbs have been highlighted in the preliminary segments shown in column 2.

These are preliminary segments because there is a significant amount of ambiguity in making precise instructions out of them. There are many sources of ambiguity. A naive decomposition of a sentence on the basis of conjunctions may not always give valid segments (*e.g.,* `enter login guest and password guest` cannot be broken into `enter login guest` and `password guest`). Within a segment, there may be multiple parses possible, making it difficult to identify the action and the targets with certainty. Column 3 shows a partial list of the possibilities that must be considered corresponding to each of the four preliminary segments, where each possibility is a candidate tuple of action, target, and the associated data. A human would resolve these ambiguities intuitively, but a machine cannot.

Even if the correct segment is selected from these many possibilities, further difficulties lie ahead in mechanical interpretation. Segments may be spurious, in that they correspond to no operation that needs to be performed. For example, the segment `Drop down list` in Figure 3 is in fact spurious; the desired work has already been done in the previous segment. Also, the target of an action on a web page may itself be ambiguous (*e.g.,* if there are multiple buttons labeled `OK` on the page).

Our main idea is to model a sequence of ambiguous test steps by a non-deterministic program, in which all possible alternative meanings of each test step have been captured. Some determinisation of this non-deterministic program is the correct sequence of desired test steps. The problem then is reduced to one of finding the correct determinisation of the non-deterministic program. Our approach accomplishes this by backtracking exploration, while trying to interpret the test steps against the application. The application acts as the test oracle. We proceed by exploring an alternative until either the entire test case can be interpreted or the interpretation reaches a point beyond which it cannot proceed. (Such an event happens if, for example, the requested action cannot be performed on the application in its current state). In the latter case, the interpretation *backtracks* to a previous decision point, and explores a different alternative. In this manner, the approach explores different determinisations of the non-deterministic program, until a successful one is found. If our tool terminates successfully, it outputs a sequence of (action, target, data) tuples that correspond to the correct determinisation. The sequence of these tuples is the automatically executable test script for that test case.

We can look at the problem of automating test automation as an instance of program synthesis [3], where the goal is to discover a program that realizes a given user intent. In our case, the user intent is stated in stylized natural language, as a sequence of manual test steps, and the goal is to synthesize a mechanically interpretable test script. Our work is also an instance of "end-user" programming [4], where the intent is to bridge the gap between natural-language and programming-language expressions, and lower the barriers to programming for end-users and novice programmers.

As a practical matter, ATA has to be prepared for incompleteness and errors in manual test steps. When it encounters a situation that it cannot resolve automatically, it stops and waits for human input. In practice, this occurs infrequently, and moreover, ATA remembers the human feedback and does not require intervention if a similar situation arises later.

C. Overview of Results

We implemented the approach and conducted three studies. Our first study measures the success of the approach in creating, with minimal human intervention, the keyword-based representation (as in Figure 2) for a suite of 33 end-to-end manual test cases for two open-source web applications and a proprietary enterprise application. These tests were written by professional testers, who were unfamiliar with our approach and the tool that implements the approach. Overall, our approach was able to automate, with no human intervention, 82% of the test steps in these test cases; the remaining steps required some user feedback.

In the second study, we evaluated the effectiveness of three optimization techniques that we designed to make the backtracking exploration more efficient. Our results indicate that the optimizations are highly effective in reducing the amount of backtracking performed, and the time required, to automate tests.

In the final study, we studied the success of segmentation on a large proprietary test corpus containing over a thousand test steps. Our results show that, in real test corpses, about half of the individual test steps contained multiple segments, and across all these segments, the tool was able to reconstruct them with 88% precision and 98% recall, supporting our hypothesis that real-world tests steps are written in a stylized manner.

D. Contributions

The key contributions of the work are the following:

1) A novel and effective approach, based on backtracking and exploration of alternative flows, for automating manual tests.
2) Three optimization techniques that significantly improve the efficiency of the technique without compromising on its effectiveness.
3) An evaluation of our approach on two publicly available web applications and three proprietary applications, on manual tests created by *professional* testers.

II. Our Approach

Our goal is to infer, from a manual test case, a sequence of automatically interpretable *action-target-data* (*ATD*) tuples, where each tuple is a triple (a, t, d) consisting of an action, a target user-interface (UI) element, and a data value.

The key intuition underlying our technique is that the presence of ambiguities gives rise to alternative interpretations (of a test step) that have to be explored. To achieve this, our technique converts the manual test case into a non-deterministic program that encodes all alternative interpretations of all test steps. This program is represented in a specialized control-flow graph, called a *test-flow graph* that contains non-deterministic decision nodes. Thus, our technique transforms the automation problem into a synthesis problem of searching for a feasible path from the root node of the test-flow graph to a leaf node (along which the entire test can be interpreted). We identify such a feasible path by exhaustively exploring the graph in a depth-first traversal, and backtracking to the last decision point when the analysis reaches a node beyond which it cannot proceed.

While exploring a new alternative at a decision node, our technique has to restore the *state* of the application to the previous state at that decision node. For web applications, the state includes the web page of the application, the values of the UI elements on that web page, the values of the session variables, and the persistent state of the back-end database. Performing state checkpoint and restoration at each decision point can be expensive. The alternative is to start each backtracking exploration from the beginning, like existing web-crawling techniques do (*e.g.,* [5], [6]); our current tool adopts this approach.

A naive approach of exhaustive exploration may be unduly expensive. Therefore, our technique uses different optimizations, which enable it to reduce the search space.

Next, we introduce some terminology. Then, we present the exploration algorithm and the search optimizations.

A. Terminology

A *manual test case* is a sequence of *test steps*, where each test step can be interpreted as a list of *segments* or a *segment list*. A test step can have multiple segment lists. We use conjunction words, such as `and`, to split a test step into segments. For instance, consider step 4 in Figure 1. Splitting on the first occurrence of `and` results in the following segment list:

```
Segment 1: enter login guest
Segment 2: password guest, and click login
```

Algorithm 1: The Algorithm for generating a test script from a test case.

```
Input: Testcase $tcase$
Output: Testscript $tscript$ or null
1  while true do
2      Invoke ExplorePath algorithm;
3      if successful then
4          Collect generated test script $tscript$;
5          return $tscript$;
       /* Choose a new path in the test-flow graph */ ;
6      Get the last decision point;
7      if last decision point has more alternatives then
8          Move to next alternative and goto Step 1;
9      if not last decision point is root node then
10         Set previous decision point as last decision point;
11         Goto step 7;
       /* Explored all paths in the test-flow graph */ ;
12     return null;
```

Algorithm 2: Algorithm `ExplorePath` (EP) for exploring a path in the test-flow graph.

```
Input: Testcase $tcase$
Output: Success or Fail
1  foreach test step $tstep \in tcase$ do
2      Compute segment lists for $tstep$ (if not already computed);
3      Get current segment list of $tstep$;
4      foreach segment $seg \in$ segment list do
5          Generate tagged segments for segment $seg$;
6          Get current tagged segment $tseg$ of $seg$;
7          Generate tuple list $tuplist$ for $tseg$;
8          if $tuplist$ list is empty then
9              return FAIL
10         foreach tuple $tle \in tuplist$ do
11             Identify UI element list for target $t$ of $tle$;
12             if element list is not empty then
13                 Get current element $uelem$ of $tle$;
14                 Execute action $a$ of $tle$ on $uelem$;
15                 if action $a$ is not successful then
16                     return FAIL
17         if at least one tuple is successful then
18             declare segment $seg$ as successful;
19     if no segment is successful then
20         return FAIL
21 return SUCCESS
```

There are four possible segment lists for this test step. Next, a *tagged segment* is a segment in which some words are annotated with POS tags. An example tagged segment for segment 1 is `enter login/noun guest`, where `login` is tagged as a noun. Segment tagging helps address the ambiguity with English words where a word can have different POS tags depending on the context of its use. For example, in the following, the first step uses "login" as a verb, whereas the second step uses it as a noun.

```
login as administrator
enter login guest
```

There can be multiple tagged segments for a segment because, for example, a verb may refer to the label of a UI element. A *disambiguated tuple* is one in which the target t is associated with a UI element of the application. Due to ambiguities in specifying target or data, there can be multiple disambiguated tuples for a tuple.

A *test-flow graph* consists of different types of nodes to represent test steps, segments, tagged segments, disambiguated tuples, and decision points. The goal of our approach is to discover a feasible path in the test-flow graph by identifying the correct alternative at each decision point.

B. Algorithms

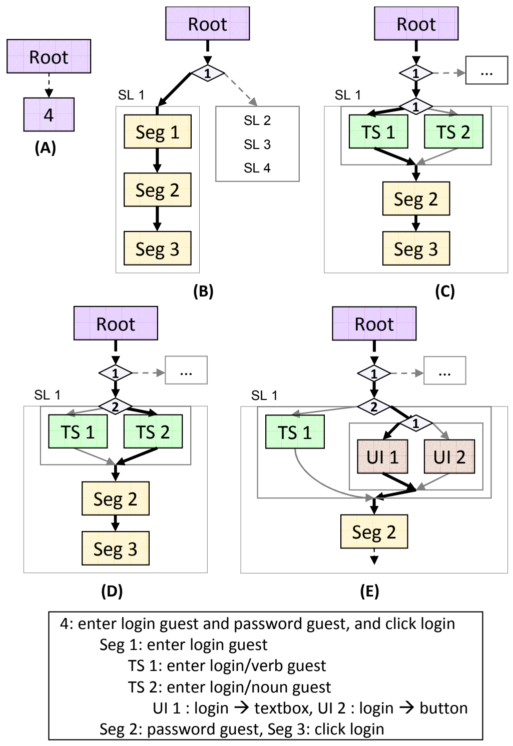
We illustrate our algorithms using illustrative examples shown in Figure 4, based on step 4 of the manual test shown in Figure 1. Algorithm 1 invokes `ExplorePath` (EP) in line 2 to explore a single path in the test-flow graph from the root node. If EP is not successful, lines 7–11 choose the next alternative for the last decision point. Algorithm 1 repeats these two steps until either a feasible path is identified or all paths are explored in the test-flow graph. We next describe the exploration of a path (Algorithm 2).

Compute segment lists (line 2): For each test step, EP identifies all segment lists by splitting the test step based on conjunction words. Currently, the implementation uses `and`, `from`, `by`, `then`, `under`, and `on` as conjunction words. If the test step includes n such phrases, the algorithm generates 2^n segment lists, representing all possible combinations of segments. For example, consider step 4 in Figure 1. Figures 4(A) and 4(B) show test-flow graphs before and after the generation of segment lists for step 4, respectively. Our algorithm constructs the test-flow graph dynamically while interpreting the test steps. Because this test step has multiple segment lists, the algorithm introduces a decision node and create a new node for each segment list. A decision point records the current alternative being explored, whose initial value is one. Figure 4(B) also shows the four segment lists (SL 1, SL 2, SL 3, and SL 4) generated for step 4. The currently explored path is highlighted with thick edges. For the sake of brevity, we do not show the details of SL 2, SL 3, and SL 4.

Generate tagged segments (line 5): The algorithm analyzes each segment in the segment list and generates a tagged segment for each segment. In particular, it identifies words that can be either nouns or verbs, and annotates those words. For example, consider Seg 1 (`enter login guest`). The algorithm generates two tagged segments (TS) as follows:

```
TS 1: enter login/verb guest
TS 2: enter login/noun guest
```

Here, `login` is tagged as a verb in the first tagged segment and as a noun in the second tagged segment. Figure 4(C)



4: enter login guest and password guest, and click login
Seg 1: enter login guest
TS 1: enter login/verb guest
TS 2: enter login/noun guest
UI 1 : login → textbox, UI 2 : login → button
Seg 2: password guest, Seg 3: click login

Figure 4. Illustrative examples for explaining the algorithm steps.

shows the test-flow graph after the tagged segments for Seg 1 have been generated. As shown, the segment node is replaced with a decision node with two successors, one for each tagged segment.

Generate tuples (lines 7–9): Next, the algorithm uses natural-language processing to generate tuples from each tagged segment. In particular, the algorithm considers the verb in a tagged segment as an action, and nouns as targets or data. Section III(A) provides more details on tuple generation. If no tuples are generated, which indicates that the algorithm had selected either a wrong tagged-segment alternative or a wrong segment-list alternative at a preceding decision point, the algorithm backtracks to choose a different alternative.

For our illustrative example, the algorithm generates no tuples for TS 1, which includes two verbs and no nouns, and therefore, does not satisfy the criteria for valid segments.[1] Therefore, the algorithm backtracks to the next alternative. Figure 4(D) shows the test-flow graph after backtracking, where the current alternative of the decision point is changed from one to two. Because TS 2 is valid, the algorithm successfully generates the tuple `<enter,login,guest>` and proceeds forward. A tagged segment can result in multiple tuples, which the algorithm processes.

Disambiguate targets (lines 11–16): The algorithm next generates disambiguated tuples by associating the target in each tuple with UI elements in the application. If there exist multiple UI elements for a target t, the algorithm introduces a new decision point, and creates a new node for each disambiguated tuple. Then, it executes each disambiguated tuple by performing the action a on the identified UI element. The action can be either one that changes the state of the application or a verification action that checks whether the application state is correct (and does not update the state). If the action fails, the algorithm backtracks to explore other alternatives.

Figure 4(E) shows the two disambiguated tuples (UI 1 and UI 2) for the tuple `<enter,login,guest>`. In the login page of the Bookstore application, "`login`" is used as the label of a text box and a button. The algorithm next performs the action and, based on the outcome of the action, either backtracks or proceeds forward.

Backtrack scenarios: As shown in Alg 2, the technique backtracks in three different scenarios:

- Segment-list backtrack (BT_{seg}), which occurs when none of the segments in a segment list is successful (line 20). Note that the algorithm does not backtrack if at least one segment in a segment list executes successfully because the remaining segments may be irrelevant and could be ignored.
- Tagged-segment backtrack (BT_{tag}), which occurs when an invalid tagged segment is encountered (line 9).
- Disambiguation backtrack (BT_{dis}), which occurs when the algorithm fails to perform an action (which may be a verification step) on the UI element of a disambiguated tuple (line 16).

Choose next alternative: Finally, when our technique backtracks, lines 7–11 of Algorithm 1 select the next alternative to explore in the test-flow graph. Selecting an alternative involves identifying the last decision point, and checking whether an unexplored alternative exists at that point. To perform this, the algorithm starts from the failing test step and iteratively checks for alternatives among disambiguated tuples, tagged segments, and segment lists. If no unexplored alternative exists, the technique moves to the preceding test step and repeats the process. If it finds an unexplored alternative, it invokes EP; otherwise, it terminates.

C. Optimizations

As mentioned earlier, our approach uses optimizations to reduce the search space in the test-flow graph. Specifically, we use three optimizations: lookahead static checking (LSC), local backtracking (LB), and active learning (AL). Each of these optimizations is safe in the sense that it can improve efficiency without compromising effectiveness: *i.e.,* an optimization cannot cause the technique to not interpret a step that it could interpret without the optimization.

Lookahead static checking (LSC): When our technique chooses a segment-list alternative for a test step, LSC statically analyzes the segment list and checks whether the

[1]A segment is considered as valid if it includes only *one* verb because each segment maps to one action performed on the application.

list includes an invalid segment. If so, LSC backtracks immediately to another alternative. The idea behind LSC is that choosing such a segment list would anyway result in a failure subsequently when the invalid segment is encountered. Therefore, LSC saves the effort that would be spent unnecessarily in exploring the segments that precede the invalid segment in the segment list.

Local backtracking (LB): Recall that, during backtracking, our technique starts exploration from the root node of the test-flow graph to address the issue of state restoration. However, at a decision point, if the application state is not modified after an alternative is chosen, it is safe to continue exploration of another alternative from that point itself—; starting again from the root node is unnecessary. To achieve this, LB monitors whether the application state is modified after an alternative is explored. LB conservatively considers the state as unmodified only when no action after a decision point succeeds.

Active learning (AL): The key idea of AL is to learn from test cases that are automated successfully, and reuse that knowledge for other test cases. To learn from successful test cases, AL identifies the alternatives chosen at each decision point of the test-flow graph of that test case. In future, during the exploration of another test case, if AL encounters a decision point with the same set of alternatives, it assigns a higher priority to those alternatives that resulted in successful interpretation of test cases previously. Note that AL does not add or delete any alternative; rather, it just assigns priorities to the alternatives, thereby, improving the efficiency without compromising effectiveness.

III. Implementation

We implemented our test-automation technique in a tool called ATA. In addition to the implementation of the backtracking algorithm, ATA consists of three main components: a component for natural-language processing, a component for runtime-interpretation, and an interface for user interaction.

A. Natural-Language Processing (NLP)

The NLP component processes a tagged segment and generates ATD tuples. First, the component annotates any unannotated word in a tagged segment using a POS Tagger [7]. Then, it uses the a natural-language parser to compute dependences among the annotated words. These dependences represent grammatical relationships between the words in the tagged segments. For example, given a tagged segment "`enter/verb login/noun`" (in which `enter` and `login` are annotated as a verb and a noun, respectively), the parser produces the dependence `dobj(enter, login)`, where `dobj` represents *direct-object dependence*. Based on this dependence, the NLP component generates ATD tuple `<enter,login,>`. The ATA NLP component includes two alternative open-source natural-language parsers: the Stanford Parser [2] and the Link Grammar Parser [8].

B. Runtime Interpretation (RI)

The RI component generates disambiguated tuples from an ATD tuple. Given a tuple, (a, t, d), RI identifies the UI elements that are candidate matches for t. To compute the candidate matches, RI extracts the textual information associated with each UI element, and constructs an *element map*, which links a UI element to a set of text labels. RI extracts different textual information for different types of UI elements. For example, for a *button*, RI extracts the *name* of the button, whereas, for a *text box*, RI extracts the name of the text box and the label preceding the text box. The reason for associating a text box with its preceding label is that manual test cases often refer to text boxes by their preceding labels. Using the element map, RI identifies all UI elements that are potential matches for t. In cases where t is not precisely mentioned in a test step (and, therefore, t does not match the text labels of any UI element in the element map), RI uses the Levenshtein edit distance [9] to identify labels that are *similar* to t. The RI component also executes the disambiguated tuples.

Currently, ATA includes three versions of the RI component, built using different testing tools: two open-source tools, HtmlUnit [10] and Selenium [11], and a commercial tool, Rational Functional Tester (RFT) [12].

C. Human-Computer Interaction (HCI)

Our technique can handle ambiguities in test steps, but it requires the test steps to be *complete*. In our context, *completeness* means that a test step should include all necessary information for mechanically interpreting the step, and that a test case should have no missing steps. However, we observed that, in practice, manual tests are often incomplete. For example, the test step `login as guest into the application` does not include all necessary information, such as password or the button to click, for signing into the application.

To address the completeness issue, ATA includes an HCI component, which lets a human to provide feedback for incomplete or missing steps. ATA highlights the failing step, and also displays the relevant web page in the application. The human can fix the test step, insert a new step, or enter tuples for the test step. ATA reuses the human-provided feedback for resolving similar test steps in other test cases as well, thereby reducing human intervention over time.

IV. Empirical Evaluation

To evaluate our technique, we conducted three empirical studies. In the first study, we evaluated the effectiveness of ATA in executing manual tests with no human intervention. In the second study, we evaluated the efficiency achieved by the three optimization techniques in reducing the time taken and the number of backtracks performed. In the

Table I
EFFECTIVENESS OF AUTOMATED INTERPRETATION OF MANUAL TESTS.

		Test Steps		Successful Steps		Modified Steps		Inserted	Reused Steps	
	Tests	Action	Verification	Action	Verification	Action	Verification	Steps	Action	Verification
BookStore	10	49	57	32 (65%)	37 (65%)	5 (10%)	6 (10%)	2	12 (24%)	14 (25%)
BugTracker	10	48	49	26 (54%)	24 (49%)	9 (18%)	15 (31%)	3	13 (27%)	10 (20%)
App1	13	62	28	42 (68%)	18 (64%)	8 (13%)	10 (36%)	0	12 (19%)	0 (0%)
Total	33	159	134	100 (63%)	79 (59%)	22(14%)	31 (23%)	5	37 (23%)	24 (18%)

third study, we evaluated the effectiveness of segmentation.[2] Following the description of the studies, we briefly present our deployment experience with ATA, and then conclude this section with a discussion of the current limitations of our technique.

A. *Effectiveness of Automated Interpretation*

1) Goals and Method: The goal of the first study is to evaluate the effectiveness of ATA in automatically interpreting natural-language test cases, with minimal human intervention. We considered the interpretation of a manual test to be successful only if ATA could execute all steps (including verification steps). We also manually verified that the automated script indeed conformed to the behavior specified in the manual test.

For this study, we used two popular open-source web applications: `BookStore` and `BugTracker` [13], and an IBM-internal enterprise application. For confidentiality, we refer to the enterprise application as `App1`. To create manual tests for the open-source applications, we identified a few scenarios that cover various independent flows in `BookStore` and `BugTracker`, and that also include manipulation of persistent data. We requested two professional testers from IBM's Testing Services to create manual tests for those scenarios. The testers were unaware of our tool and technique, and the goal of the study (*i.e.,* the evaluation of a test-automation tool). The only instructions they were given were to be elaborate in writing the tests and mention the test data wherever needed. They were given no other instructions, such as on segmentation and different ambiguities. Thus, we eliminated bias, to the extent possible, in the creation of the tests. In total, the testers wrote 23 manual tests—11 for `BookStore` and 12 for `BugTracker`—which together consist of 203 steps. For `App1`, we used 15 manual tests, consisting of 90 steps, from the existing test suite of the application.

2) Results and Analysis: Table I presents the results of the study. Overall, ATA automated 10 of the 11 tests for `BookStore`, 10 of the 12 tests for `BugTracker`, and 13 of the 15 tests for `App1`. Table I shows data for only the 33 tests that could be automated. (Later in this section, we discuss the reasons why some tests could not be automated.) Columns 3 and 4 show the number of action and verification steps, respectively, in each subject. Columns 5 and 6 show the number of steps that are automatically interpreted by ATA, *with no human intervention*. Columns 10 and 11 show the number of steps that are interpreted automatically, but by reusing human feedback obtained for other steps. Together, these columns illustrate the extent of automated interpretation achieved by ATA. For the action steps, $137/159$ (86%) could be automatically interpreted by ATA—63% with no human intervention and 23% via feedback reuse. For the verification steps, the success rate was lower: $103/134$ (77%) of the verification steps could be automatically interpreted. Over both action and verification steps, automated interpretation could be achieved for $240/293$ (82%) steps. In terms of the required human intervention, 14% of the action steps and 23% of the verification steps required human input in order to be interpreted.

At the granularity of tests, all the tests required at least minimal feedback for the login step. But, 11 of the 33 tests were automated via reused feedback only, with no human intervention.

Next, we present insights into the nature of the human feedback that ATA required.

Action Steps: Among the failing action steps, there are two main categories. First, some steps do not describe the target in the application correctly. For instance, consider the step `Click on the Bug Tracking tab`. The specified target, `Bug Tracking tab`, is actually an image in the application. Therefore, ATA could not identify any target for this step. Second, some steps do not explicitly mention the data, or refer to data created in previous steps. For example, consider the step `Click on the Employee link of the Employee that has been updated`. Currently, ATA does not perform any analysis to identify such dependences; therefore, it cannot execute such steps. Indeed, in `BugTracker`, $4/9$ (44%) of the steps that required modifications (column 7) occur in one test case only.

Verification Steps: We have found that, in general, verification steps are written in a more informal manner, with less structure, than action steps. Therefore, verification steps pose a greater challenge for automated interpretation, as illustrated by our empirical data: $15/49$ (31%) of the verification steps in `BugTracker` required modifications.

Among the failing verification steps, there are two main categories. First, some steps do not describe precisely what has to be verified. For example, consider the following verification step from `BookStore`: `User should be able To Cancel the updation to the user profiles`. This step does not clearly state what has to be checked. If the step

[2]The complete dataset including the manual tests and the generated scripts is available at http://tinyurl.com/83ehhds (also available on request from the authors)

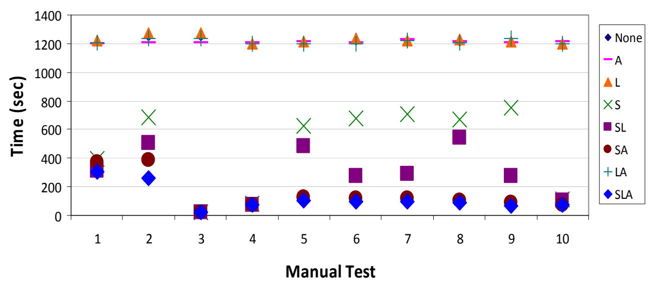


Figure 5. Time taken for automation using optimizations.

were to say what page should be shown after the action, ATA would have automated the step. Second, some verification steps require logic to be coded. For instance, consider the step: `The Bug names in the Bug column should be rearranged to be displayed in sorted order`. Such verification steps require the coding of custom logic, which cannot be automatically synthesized. To address such cases, ATA has a feature, which lets users specify the name of a custom function as the "action" of an ATD tuple; the specified function is automatically invoked by ATA during test interpretation. More generally, users can create a library of custom functions (possibly, using a testing tool such as RFT), and link the library to ATA.

Inserted Steps: Among the missing steps, the most common case is related to signing into the application. For example, many tests required the insertion of the following step: `Enter login guest and password guest, and click login`. Column 9 of Table I shows the number of distinct steps that were added to the manual tests. Our feedback mechanism has indeed a valuable side-benefit—while performing automation, testers can also improve the manual tests and keep them up-to-date.

For `BookStore` and `BugTracker`, three tests could not be automated due to lack of support for state restoration of persistent database in our current prototype. Section IV-E discusses this issue further. For `App1`, two tests could not be automated, one of which required looping over test steps.

B. Benefits of the Optimization Techniques

1) Goals and Method: The goal of the second study was to evaluate the efficiency of the three optimization techniques in reducing the time taken for automation and the number of backtracks performed. We use S, L, and A to refer to lookahead static checking, local backtracking, and active learning optimizations, respectively. For this study, we automated the 10 (successful) manual tests of `BookStore` using 8 different configurations: with all optimizations disabled (1 configuration) , with each optimization enabled individually (3 configurations), with pairs of optimizations enabled (3 configurations), and with all optimizations enabled (1 configuration). In each configuration, we ran ATA for a maximum time of 20 minutes for each manual test. If ATA does not complete automation in 20 minutes, it records

Table II
EVALUATION RESULTS FOR METRICS: BT_{seg}, BT_{tag}, AND BT_{dis}.

	BT_{seg}	BT_{tag}	BT_{dis}
S	2063	9	24
SL	2091	9	24
SA	1631	1	7
SLA	1639	1	7

Table III
ACCURACY OF SEGMENTATION.

Subject	Test Cases	Test Steps	Test Steps with Multiple Segments	AS	RS	$AS \cap RS$	Precision	Recall
App2	121	648	325 (50%)	675	682	664	97%	98%
App3	46	426	120 (28%)	258	362	250	69%	97%
Total	167	1074	445 (41%)	933	1044	914	88%	98%

the result as a failure and proceeds to the next manual test. We used four measures in our study: time taken, number of BT_{seg}, BT_{tag}, and BT_{dis} (Section II-B). All experiments were conducted on an Intel Core 2 Duo CPU machine with 2.53 GHz and 3 GB RAM.

2) Results and Analysis: Figures 5 shows the time taken (in seconds) using the eight configurations. The result shows that, without static checking, the other configurations (L, A, and LA) could not finish within the allotted time, which illustrates that static checking is a very important optimization. Among the other configurations, local backtracking helped SLA perform better than SA for the initial two manual tests. However, after ATA gained knowledge—through active learning—of the right alternatives at different decision points, SLA and SA took similar time.

Table II shows the backtracking data for configurations: S, SL, SA, and SLA. We do not show the results of "none", L, A, and LA configurations because they could not complete within the allotted time. Overall, the data show that active learning significantly helped reduce the number of backtracks.

In summary, static checking helps improve efficiency significantly by reducing the time that would be spent unnecessarily in exploring the segments that precede an invalid segment in a segment list. Local backtracking helps when ATA encounters new decision points, whereas active learning helps to choose intelligently the right alternatives at decision points, based on the knowledge gained from automating the previous manual tests.

C. Accuracy of Segmentation

1) Goals and Method: The goal of the final study was to perform a more extensive evaluation—over a much larger corpus of manual tests—the effectiveness of segmentation, on which the backtracking technique is based. This study also provides evidence that, in practice, test steps are often composed of multiple segments. For this study, we used two enterprise applications maintained by the IBM Testing Services practice; we refer to the applications as `App2` and `App3`. We used a suite of the manual tests written for these applications. For lack of space, we present this study briefly here; the detailed results are available in Reference [14].

To evaluate the accuracy of segmentation, we measure *precision* and *recall* of retrieving segments from the set of manual steps.

2) Results and Analysis: Table III shows our results. Here, AS and RS represent the segments that can be potentially derived and the segments retrieved by ATA, respectively. The results show that a large fraction (50% in `App2` and 28% in `App3`) of manual test steps require to be split into segments before an automation task becomes meaningful. On average, 87% of the 445 steps that needed segmentation are composed of just two segments; but, there are steps that contain as many as six segments. For `App2`, both precision (97%) and recall (98%) are very high. The cases for `App2` are complex—many sentences are long-winding, with a wide variety of constructions of clauses. Consequently, many spurious segments are identified and precision drops to 69%, even as recall remains high at 97%.

D. Benefits Observed in a Production Environment

We have also evaluated ATA in a production environment to assess it usefulness in improving tester productivity in automation. We deployed ATA in an IBM-internal project in a pilot evaluation conducted over an eight-week period, involving four testers. Prior to using ATA, this team had automated about 900 manual tests over a period of 10 months, via writing Java code in RFT, with the average automation rate of 3–5 tests per day per tester. Using ATA, the testers automated close to 600 tests in eight weeks, with the average of 10 tests per day per tester; thus, ATA *doubled the tester productivity* in automating manual tests. In addition, we found that ATA is quite easy to use and does not require a high level of expertise in programming or proficiency in automation tools, such as RFT.

E. Discussion

Our results indicate that the backtracking-based approach is quite effective in being able to interpret manual tests, even in the presence of different types of ambiguities. However, there are limitations of the current technique, which we discuss here.

Multiple feasible executions: It is possible that there could be multiple feasible paths in the test-flow graph. Currently, ATA computes the automated script using the first feasible path that it finds. However, ATA could be easily adapted to generate all feasible paths by not terminating, but simply continuing further after the first feasible path is found, and leaving the resolution of the correct interpretation, among the candidate feasible interpretations, to human judgment.

State restoration: While exploring a new alternative during backtracking, ATA restores the *state* of the application to the previous state at each decision node. Currently, ATA supports only application-level state restoration—it cannot restore the persistent state of the back-end database. In our evaluation, ATA could not automate three manual tests (one in `BookStore` and two `BugTracker`) because of this limitation. In future work, we plan to address this limitation by using database-level checkpoint and rollback to restore the persistent state.

Correctness of the ATA-generated script: ATA generates a test script *only* if all verification steps pass, which gives high confidence in the correctness of the generated script. However, we recommend the user of ATA to verify the final script by playing back the script, to guard against the rare probability that ATA chooses an incorrect feasible path that also passes all verification steps.

V. Related Work

Conventional approaches for test automation include record-replay and keyword-driven automation, discussed in the Introduction. Record-replay, which is a feature available in many commercial and open-source tools (*e.g.,* RFT [12] and Selenium [11]), requires a human to perform the manual test steps on the application user interface. Keyword-driven automation involves the creation of a library of reusable subroutines or *keywords*. A manual test case is translated (by a human) into a sequence of keywords; a driver program interprets such sequences by invoking appropriate subroutines. Both these techniques require human interpretation of English language tests, whereas our approach attempts to eliminate this and can work in a more unattended manner.

There is a large body of work on synthesizing programs via mechanical interpretation of natural-language phrases, which is inspired by the ideal of bringing programming to end-users and bridging the gap between human-readable natural languages and mechanically interpretable programming languages [15]. A recent symposium on the future of software-engineering research discussed the automated generation of formal software-engineering artifacts, such as models and test cases, from natural-language descriptions as one of the important research directions [16].

The most notable effort in the direction of end-user programming, with a long line of research, is programming by demonstration [4], [17]—which expounds the principle that a program is synthesized automatically by observing the manual actions performed by the user. Other efforts at bridging the gap between natural languages and programming languages include the development natural-language interfaces for programming languages (*e.g.,* [18]), providing semantics to natural-language interfaces (*e.g.,* [19]), and the development of structured editors [15].

Automated program synthesis has been studied more broadly as the problem of discovering a program that realizes a user intent—the intent could be stated in different forms such as, natural language, input-output examples, logical relations between inputs and outputs, and demonstrations; the users could range from algorithm designers and programmers to students and end-users [3]. Gulwani [3] describes three dimensions of the synthesis problem: the form of user intent, the search space of programs, and the search technique. Our work can be formulated within this framework. We target intent stated as natural-language imperative-form commands. The search space consists of tuples of keywords (from the keyword domain of a specific testing framework) and target elements (from the domain of labels of user-interface elements used in the application under test). The search technique is based on natural-language parsing combined with exploration of multiple flows via backtracking.

Recent approaches in automated program synthesis consider user intent specified in different forms, such as input-output mappings (*e.g.,* [20]), logical relations between inputs and outputs (*e.g.,* [21]), and traces (*e.g.,* [22]).

Closer to our work, techniques for automatically interpreting user intent stated in stylized natural-language expressions have been developed in the contexts of test scripting [23], synthesizing code from informal keyword expressions [24], automating web-based processes [25], [26], and guiding users in following "how-to" instructions [27].

The CoTester system [23] uses an English-like test-scripting language, called ClearScript, which can be automatically interpreted. CoTester provides a record-replay feature similar to that available in testing tools, with the difference that the recorded scripts are in the stylized English form of ClearScript and, therefore, easily readable even by non-programmers. Although ClearScript test steps are expressed in restricted English language, each step must correspond precisely to a segment. Thus, the burden of segmentation is on the user. Our approach removes this restriction. In essence, programming in ClearScript is at the level of keyword-driven approaches, albeit with a more user-friendly natural-language syntax; the higher level of informality is more in the syntax, and not in degree of precision in specifying test steps. Moreover, the target-ambiguity problem is not addressed in CoTester.

The test-scripting language of CoTester, ClearScript, was developed in earlier work in the context of automating web-based processes (*e.g.,* expense reimbursement) in enterprises [25], [26]. The ClearScript specifications require accurately segmented and ordered process steps.

Recently, Lau and colleagues [27] have developed techniques for automated interpretation of "How-to" instructions to provide guidance to end-users in accomplishing web-based tasks, such as making an online purchase or creating a web-based email account. They formalize the interpretation problem as the inference of a tuple (A, V, T) consisting of an action, a data value, and a target element; the inference of such a specification is our goal too. Their approach targets instructions written by end-users, which can have a greater degree of informality and ambiguity than what would typically be present in the manual tests targeted by our approach. However, they assume that the instructions have been properly segmented and, therefore, do not handle segmentation ambiguity. They also do not handle target ambiguity.

Little and Miller [24] present a technique for automatically translating keywords, *i.e.,* informal plain-text expressions, to the API of a particular system. The goal of their work is to lower the barriers for end-users in leveraging the scripting interfaces provided by applications such as Microsoft Word. The users can specify a task, such as formatting a document, using plain-text keywords, which the system attempts to translate to the application API. In this approach, an expression corresponds roughly to a segmented action. This approach is similar in principle to API-discovery techniques, such as Jungloid mining [28], but requires less formal expressions that are more appropriate for end-users.

In related work, researchers have presented techniques for analyzing natural-language descriptions of use cases to extract models, measure quality, detect defects, and generate test cases (*e.g.,* [29], [30], [31]).

The Cucumber tool [32] lets functional descriptions or scenarios to be written in a language called Gherkin (a business readable domain-specific language), which is English-like in flavor, but with an underlying syntax. Such scenarios can be automatically executed by writing the "glue" code for a particular testing tool/framework (*e.g.,* Selenium or JUnit). In contrast, our approach interprets manual tests written in unconstrained natural language with no underlying syntax, and attempts to resolve different types of ambiguities.

VI. Conclusion

We presented a technique using which manual tests written in English can be almost automatically converted to a form ready for mechanical interpretation. The technique uses backtracking-based search to resolve ambiguities inherent in the tester's instructions. The importance of this work is that it can drastically reduce the cost of test automation. Test automation is a necessary cost in testing practice because manual tests cannot be executed repeatedly in an efficient manner; they have to be converted to executable scripts. Our work automates the conversion to script form. The technique accomplishes this with remarkably high accuracy—automating 82% of 293 steps in our corpus of test cases.

In future work, we will further evaluate the technique in production environments, add support for persistent state restoration, and investigate whether our technique can be combined with Text2Test [31] to enable automation from natural-language use cases to executable test scripts.

Acknowledgments

We thank Achin K Das and Srinivas Mangipudi for their discussions in identifying the test-automation problem. We also thank Debdoot Mukherjee for helping with the experiments, and the anonymous reviewers for their detailed and insightful comments.

References

[1] "Junit." [Online]. Available: http://junit.org

[2] M.-C. D. Marneffe, B. Maccartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *Proc. of the LREC*, 2006, pp. 449–454.

[3] S. Gulwani, "Dimensions in program synthesis," in *Proc. of PPDP*, 2010, pp. 13–24.

[4] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, Eds., *Watch what I do: Programming by demonstration*. Cambridge, MA, USA: MIT Press, 1993.

[5] A. Mesbah, E. Bozdag, and A. v. Deursen, "Crawling AJAX by inferring user interface state changes," in *Proc. of ICWE*, 2008, pp. 122–134.

[6] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence, "A combinatorial approach to building navigation graphs for dynamic web applications," in *Proc. of ICSM*, 2009, pp. 211–220.

[7] "Stanford log-linear part-of-speech tagger." [Online]. Available: http://nlp.stanford.edu/software/tagger.shtml

[8] "Link grammar." [Online]. Available: http://www.link.cs.cmu.edu/link/

[9] "The levenshtein-algorithm." [Online]. Available: http://www.levenshtein.net/

[10] "Html unit." [Online]. Available: http://htmlunit.sourceforge.net/

[11] "Selenium web browser automation." [Online]. Available: http://seleniumhq

[12] "IBM rational functional tester." [Online]. Available: http://www-01.ibm.com/software/awdtools/tester/functional/

[13] "Goto code." [Online]. Available: www.gotocode.com

[14] S. Thummalapenta, S. Sinha, D. Mukherjee, and S. Chandra, "Automating test automation," IBM Research, Tech. Rep. RI11014, 2011.

[15] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, pp. 83–137, June 2005.

[16] W. F. Tichy and S. J. Koerner, "Text to software: Developing tools to close the gaps in software engineering," in *Proc. of FOSER*, 2010, pp. 379–384.

[17] H. Lieberman, Ed., *Your wish is my command: Programming by example*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[18] D. Price, E. Riloff, J. Zachary, and B. Harvey, "NaturalJava: A natural language interface for programming in Java," in *Proc. of IUI*, 2000, pp. 207–211.

[19] H. Liu and H. Lieberman, "Programmatic semantics for natural language interfaces," in *Proc. of CHI Extended Abstracts on Human Factors in Computing Systems*, 2005, pp. 1597–1600.

[20] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. of ICSE*, 2010, pp. 215–224.

[21] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proc. of POPL*, 2010, pp. 313–326.

[22] T. Lau, P. Domingos, and D. S. Weld, "Learning programs from traces using version space algebra," in *Proc. of K-CAP*, 2003, pp. 36–43.

[23] J. Mahmud and T. Lau, "Lowering the barriers to website testing with CoTester," in *Proc. of IUI*, 2010, pp. 169–178.

[24] G. Little and R. C. Miller, "Translating keyword commands into executable code," in *Proc. of UIST*, 2006, pp. 135–144.

[25] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Co-scripter: Automating and sharing how-to knowledge in the enterprise," in *Proc. of CHI*, 2009, pp. 1719–1728.

[26] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: capture, share, automate, personalize business processes on the web," in *Proc. of CHI*, 2007, pp. 943–946.

[27] T. Lau, C. Drews, and J. Nichols, "Interpreting written how-to instructions," in *Proc. of IJCAI*, 2009, pp. 1433–1438.

[28] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proc. of PLDI*, 2005, pp. 48–61.

[29] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, "Application of linguistic techniques for use case analysis," in *Proc. of RE*, 2002, pp. 157–164.

[30] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev, "A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases," in *Proc. of DSN*, 2009, pp. 327–336.

[31] A. Sinha, S. Sutton, and A. Paradkar, "Text2Test: Automated inspection of natural language use cases," in *Proc. of ICST*, 2010, pp. 155–164.

[32] "Cucumber." [Online]. Available: http://cukes.info/